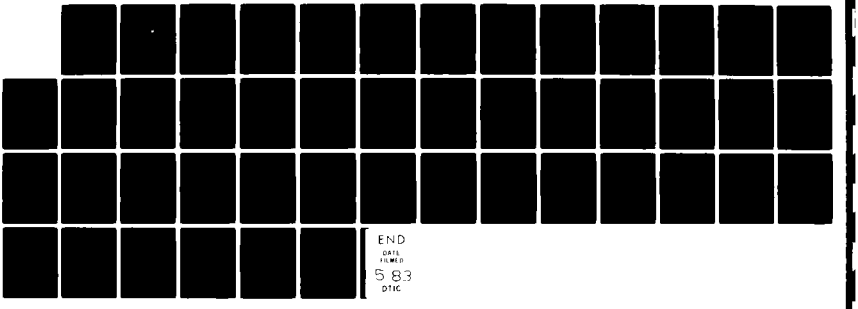
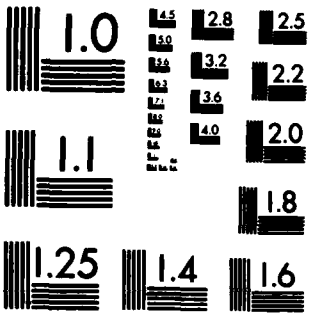
AD-A127 061 ORGANIZATIONAL STRUCTURE CONSIDERATIONS FOR SOFTWARE DEVELOPMENT PROJECTS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA K M QUINN DEC 82 1/1

UNCLASSIFIED F/G 5/1 • NL

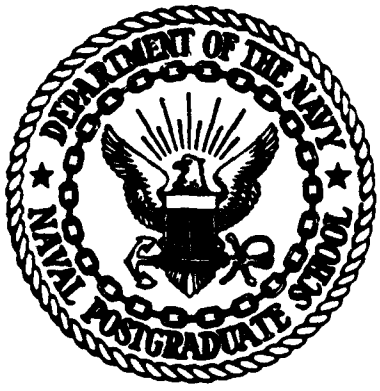




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(2)

NAVAL POSTGRADUATE SCHOOL
Monterey, California



AD-A127 [illegible]

THESIS

1983

A

ORGANIZATIONAL STRUCTURE CONSIDERATIONS FOR SOFTWARE DEVELOPMENT PROJECTS

by

KEVIN M. QUINN

December, 1982

Thesis Advisor Norman Lyons

Approved for Public Release, Distribution Unlimited

DTIC FILE COPY

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. A127061	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Organizational Structure Considerations for Software Development Projects		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December, 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kevin M. Quinn		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA		12. REPORT DATE December, 1982
		13. NUMBER OF PAGES 45
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release, Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Organizational Structure, Software Development
Team Size, Standardization, Division of Work

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Organizational structure has long been recognized as having an important impact on an organization's ability to accomplish its objectives. This paper provides managers of software development projects with an analysis of the importance of several elements of organizational structure, and of how they can use this knowledge to make decisions which will have a positive impact

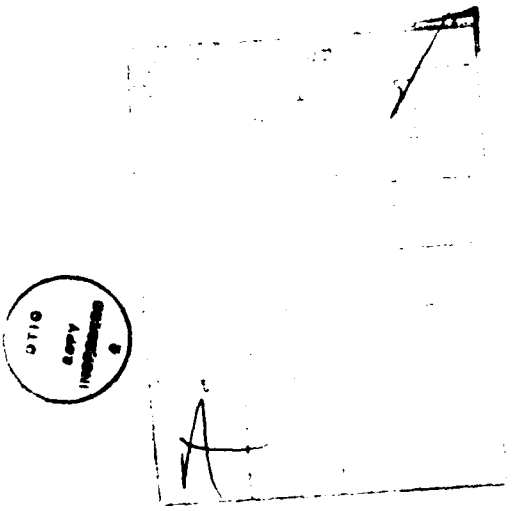
(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued) Block # 20

on the success of their projects. The structural elements discussed are specialization of activities, size of the work group, and standardization of activities.



Approved for public release; distribution unlimited.

Organizational Structure Considerations for Software Development Projects

by

Kevin M. Quinn
Lieutenant, United States Navy
B.S., United States Naval Academy, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
December 1982

Author: Kevin M. Quinn

Approved by: Norman R. Lyons
Thesis Advisor

Barry D. Nunn
Second Reader

Chairman, Department of Administrative Sciences

W. M. Woods
Dean of Information and Policy Sciences

ABSTRACT

Organizational structure has long been recognized as having an important impact on an organization's ability to accomplish its objectives. This paper provides managers of software development projects with an analysis of the importance of several elements of organizational structure, and of how they can use this knowledge to make decisions which will have a positive impact on the success of their projects. The structural elements discussed are specialization of activities, size of the work group, and standardization of activities.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

As software development projects have become more and more complex, organizations have developed various structures to accomplish them in an effective, efficient, and timely manner. This aspect of the organizational adaptation process involves manipulating elements of organizational structure in such a way as to optimize the utilization of the organization's scarce resources. Stoner [Ref. 1] identifies four major determinants of organizational structure: the organization's strategy for achieving its goals; the skills and needs of the people employed; the technology employed; and the size of the organization and its subunits. Management, by making decisions concerning these determinants, seeks to develop the structure which will be most successful in accomplishing the goals of the organization. These managerial decisions are of extreme importance because "the choices which top management make are the critical determinants of organizational structure and process." [Ref. 2: p.548] Because of the impact of these decisions, it is very important that managers of a software development project understand what the elements of organizational structure are and how they can be manipulated to improve the performance of the development process.

Three elements of organizational structure noted by Stoner [Ref. 1] will be the focus of this paper. The first element will be the specialization of activities. This concerns the breaking down of the overall project into component activities and assigning personnel with specialized training to accomplish those activities for which their training makes them most suited, and in which they will be most productive. The objective of the chapter on

specialization of activities will be the making the critical decisions of the optimal combination of specialized labor to employ, and the optimal quantity of individual specialized labor to apply to the software development process.

The second element will be the size of the work group. An analysis of the impact of work group size on productivity will be made. The factors which influence work group productivity will be explored, and approaches to mitigating the negative factors while enhancing the positive factors will be examined. The size and composition of a work group with high productivity potential will be investigated, and the managerial and systems design techniques needed to support this work group will also be noted.

The third and final element will be the standardization of activities. The benefits of activity standardization within a software development project will first be discussed in general terms. The standardization of one phase of the process will then be analyzed in detail to identify specific contributions and relevance to the overall management process.

II. SPECIALIZATION OF ACTIVITIES

A. REASONS FOR SPECIALIZATION

One of the most important elements of organizational structure is the specialization of activities. This specialization in the organizational sense includes the breaking down of the project into smaller, specialized tasks. The benefits of division of work have been repeatedly demonstrated throughout the history of civilization. The order of magnitude improvements in productivity resulting from division of work have had profound impact on the world's industrial development. Division of work is important because

> no one person is physically able to perform all of the operations in most complex tasks, nor can any one person acquire all the skills needed to perform the various tasks that make up a complex operation. Thus, in order to carry out tasks requiring a number of steps, it is necessary to parcel out the various parts of the task among a number of people. Such specialized division of work allows people to learn skills and become expert at their individual job functions. Simplified tasks can be learned in a relatively short period of time and be completed quickly. [Ref. 1: p. 254]

In a complex task such as a software development project it would be impractical to assign one person to accomplish the task by himself. In order to achieve a high quality product, this person would not only have to be expert in all areas of software development, he would also have to be able to provide his own clerical services, administative services, computer services, etc. This one-man approach is impractical for a multitude of reasons, not the least of which is the development time that would be required. With development times for projects running into the hundreds or

thousands of man-years, only the smallest of projects would be possible. Therefore, division of work in a complex development project is absolutely essential to the success of the project.

B. SPECIALIZATION IN SOFTWARE PROJECTS

The software development project is often broken down into a sequence of tasks, phases, or activities such as requirements analysis, system design, system coding, system test, etc. This division of the overall task into many subtasks has been widely discussed in the literature.

As the task itself is divided into a variety of subtasks, so to must the overall work requirement be divided among many individuals. As discussed above, this division of labor is necessary to produce a quality product within time and cost constraints. The division of labor allows individuals to specialize and become expert at certain skills. Systems analysts, programmers, technical analysts, and database administrators are some of the specializations within software development projects. The chief programmer team concept as described by Brooks [Ref. 3: p. 32-35], makes clear distinctions between the skills, duties, and responsibilities of its team members. The specialization within the chief programmer team includes a chief programmer, assistant programmer, administrator, editor, secretaries, clerk, toolsmith, tester, and language lawyer. This type of team, the individuals within it and their duties will be discussed later in the paper.

C. MANAGEMENT DECISIONS

The thrust of this chapter will be towards developing generic conceptual frameworks for the management decisions concerning the combination and quantity of the specialized

labor skills to employ. The following management decision questions will be addressed:

1) What is the optimal mix of the different types of labor to employ in the software development process?

2) What is the optimal quantity of a particular type of labor to employ?

1. The Labor Mix Decision

a. The Production Process

The software development process is a production process which transforms a particular set of inputs (e.g. systems analyst labor, programmer labor, computer services, etc.) into a desired output. Figure 2.1 models this process.

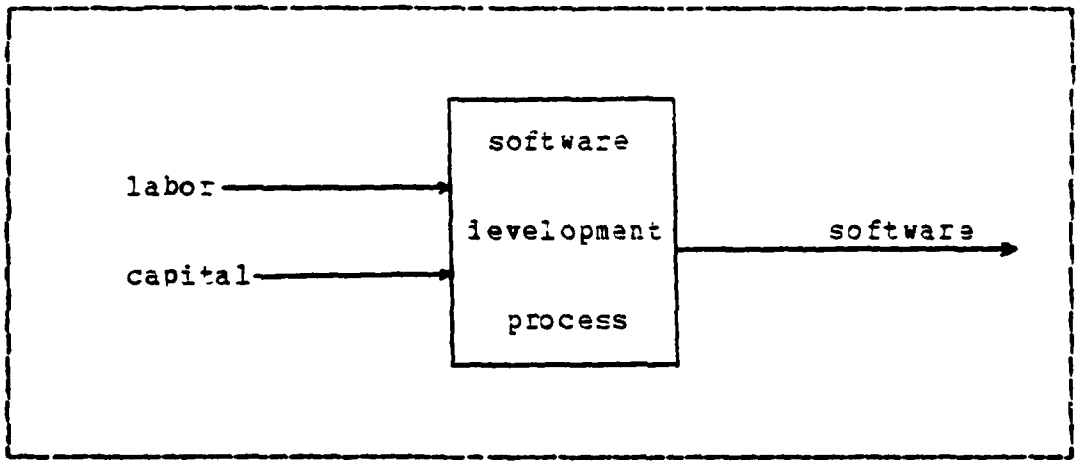


Figure 2.1 Software Development Model.

The relationship between the inputs into the process and the maximum output based upon those inputs represents the production function for the process. In other words, given the technology applied, the output of the process is a function of the inputs employed in the process. Brooks [Ref. 3] and Fried [Ref. 4] have demonstrated that if the other inputs are held constant while one type of labor is allowed to increase, that input will, at some point, show decreasing marginal productivity and will eventually display a negative marginal productivity. Figure 2.2 illustrates these findings with respect to programmer labor. The slope of the curve in Figure 2.2 represents the marginal product of programmer labor with respect to total lines of code produced.

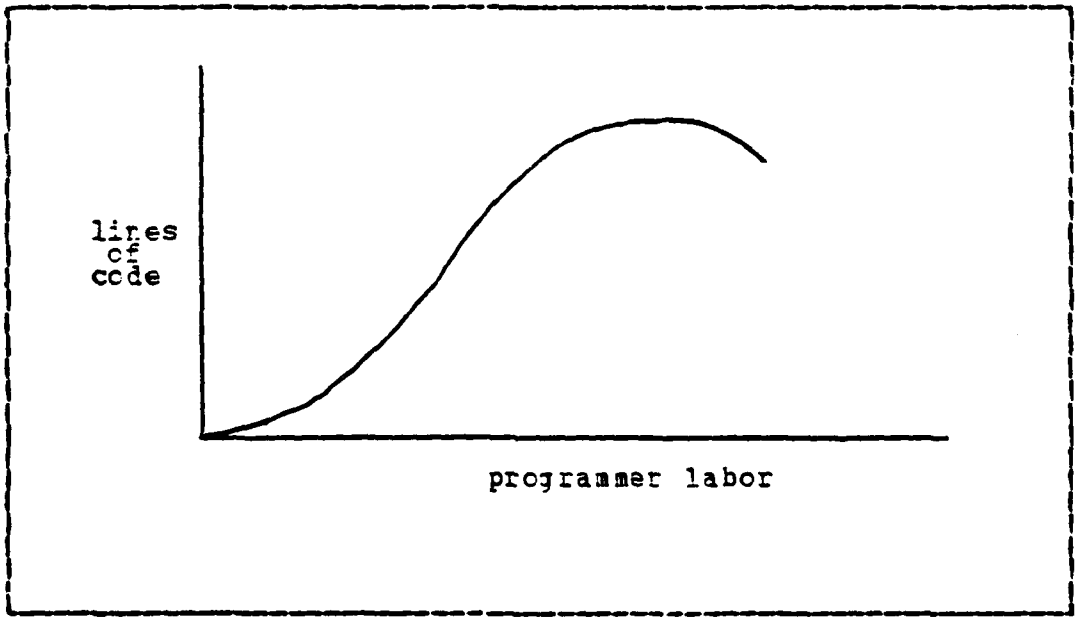


Figure 2.2 Programmer Labor Productivity.

If we allow two of the inputs to vary, we can develop an isoquant representing all of the possible efficient combinations of these two inputs which will produce the same quantity of output. For the purpose of this argument the two inputs used will be systems analyst labor and programmer labor. Figure 2.3 illustrates an isoquant which represents the possible combinations of programmer labor and systems analyst labor capable of producing a given quantity of software (Qs).

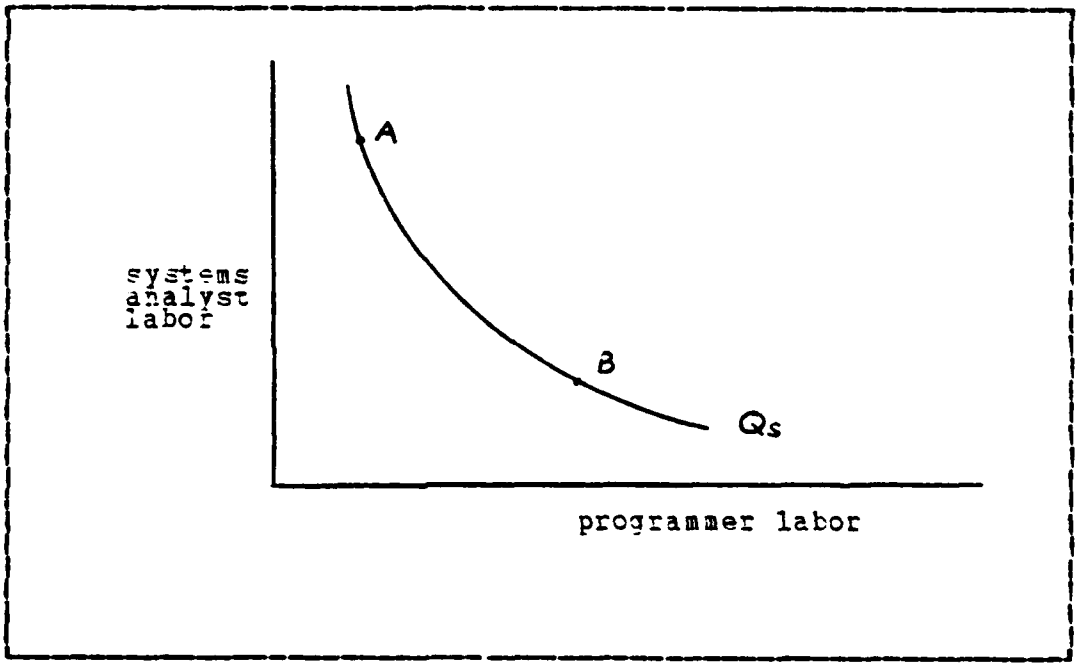


Figure 2.3 A Software Development Isoquant.

Points A and B on the isoquant represent two different combinations of programmer and systems analyst labor capable of producing the same amount of software; as such, they represent two different technological processes (within the given technology) used in the production of the

software. The slope of the curve, therefore, represents the marginal rate of technological substitution (MRTS) of programmer labor for systems analyst labor. It can also be shown that the marginal rate of technical substitution is equal to the ratio of the marginal products of the inputs [Ref. 5: p. 158]. In symbols:

$$MRTS = -MPp/MPsa \qquad \text{(eqn 2.1)}$$

where:

MPp = marginal product of programmer labor
MPsa = marginal product of systems analyst labor
MRTS = marginal rate of technical substitution.

b. The Costs

If we assume that the organization has a limited amount of funds to expend on the inputs to the production process, and that the total cost of the fixed inputs remains constant and is less than the total amount available, then there exists an amount which is available to partition among the variable inputs: programmer and systems analyst labor. In symbols:

$$M = Pp*Qp + Psa*Qsa \qquad \text{(eqn 2.2)}$$

where:

M = the total amount available for programmer and systems analyst labor
Pp = the price of a unit of programmer labor
Psa = the price of a unit of systems analyst labor
Qp = the quantity of programmer labor used
Qsa = the quantity of systems analyst labor used.

If we graph equation 2.2 we can represent the various possible combinations of programmer and systems analyst labor that can be aquired for the amount M by a straight line as in Figure 2.4 . This line is called the isocost curve for these input combinations. The slope of the isocost curve is negative and can be shown to be equal to Pp/Psa.

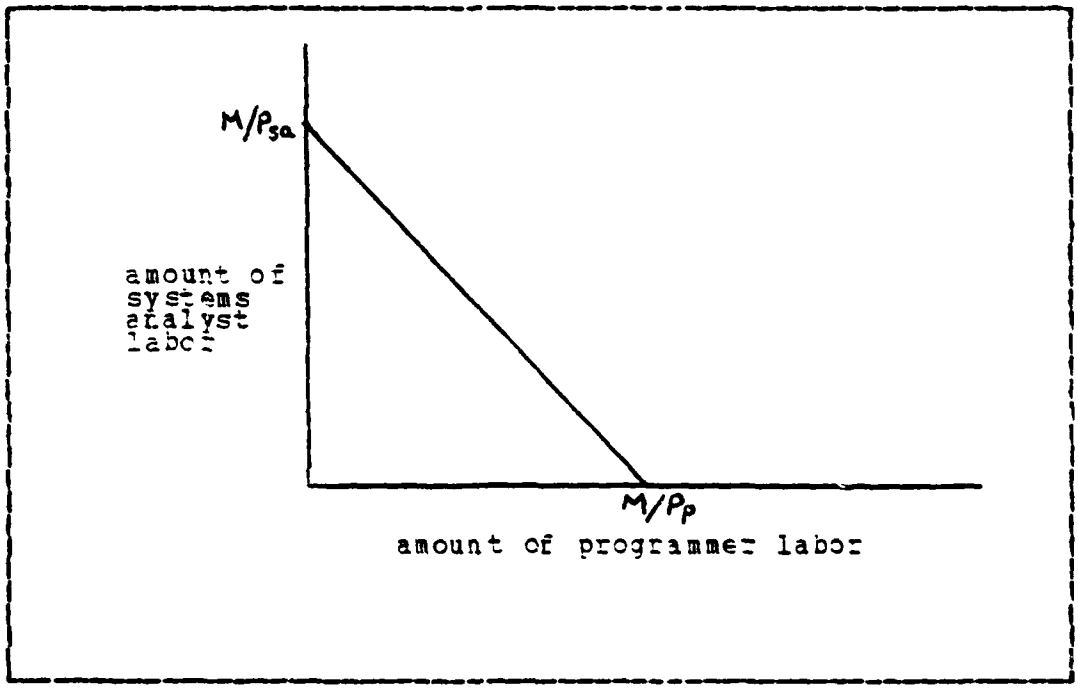


Figure 2.4 Isocost Curve.

If a family of the previously developed isoquant curves is superimposed upon the isocost curve of Figure 2.4, as in Figure 2.5, it is possible to graphically determine the optimum mix of programmer and systems analyst labor to employ in the software development process.

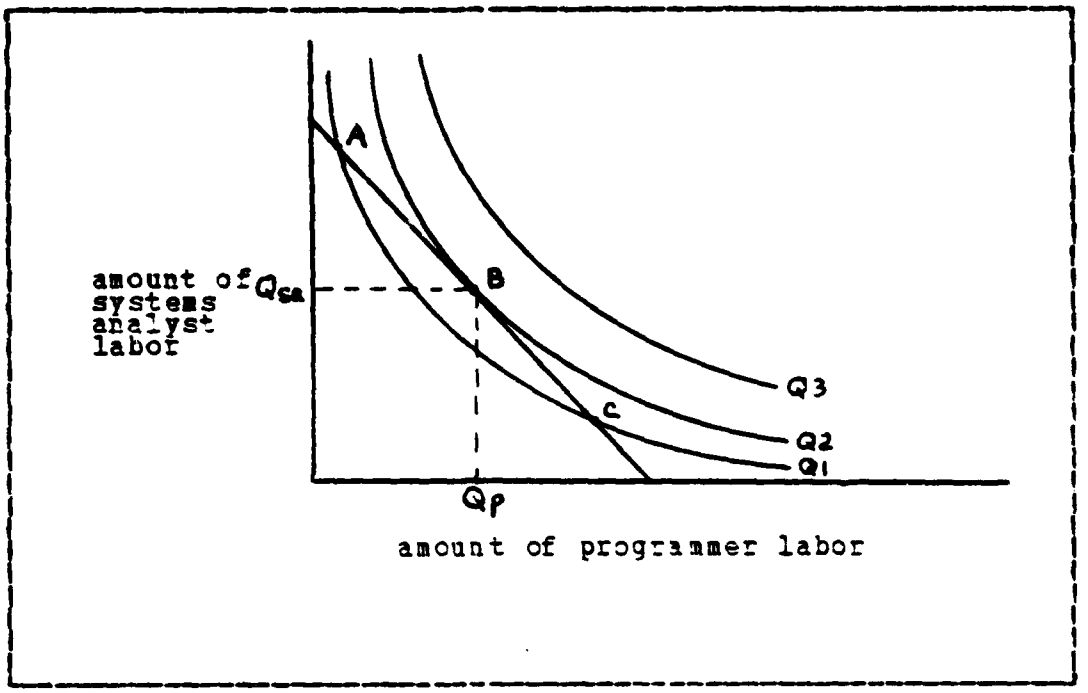


Figure 2.5 The Optimum Labor Mix.

Q1, Q2, and Q3 represent isoquants in increasing order of quantity of software produced. There may be any number of isoquants represented on the graph, but it can be seen that output will be maximized for a given cost (M) at the point where the isocost curve is tangent to the highest isoquant curve. In Figure 2.5 this is point B, and Q2 is the maximum quantity of software that can be produced for the given dollar amount available for programmer and systems analyst labor. Alternately, it could be stated that M is the minimum amount that would have to be spent on programmer and systems analyst labor, other factors held constant, in order to produce a desired quantity Q2. points A and C represent suboptimum utilization of resources because the same dollar amount is being expended to produce a smaller amount of output (Q1) than it is possible of producing.

Additionally, Figure 2.5 shows that, if other factors of production are held constant, it is not possible to produce more than Q2 (for example Q3) with a limit of M dollars available for programmer and systems analyst labor. Therefore, from Figure 2.5 it is demonstrated that the optimum mix of systems analyst and programmer labor is represented by the quantities Qsa and Qp respectively.

There is still more information available from Figure 2.5 . As shown above, the slope of the isocost curve is equal to the ratio of the prices of the inputs (-Pp/Psa), and the slope of the isoquant curve is equal to the marginal rate of technological substitution or the ratio of the marginal products of the inputs (-MPp/MPsa). The optimum mix has been shown to be the point of tangency between the isocost curve and the isoquant curve. Therefore, at the optimum, the ratio of the prices of the inputs will be equal to the ratio of their marginal products. The optimal combination of programmer and systems analyst labor, therefore, is where:

$$Pp/Psa = MPp/MPsa \qquad \text{(eqn 2.3)}$$

or alternately where:

$$MPsa/Psa = MPp/Pp \qquad \text{(eqn 2.4)}$$

This second equation reveals that the optimum mix exists where the marginal productivity of a dollar's worth of systems analyst labor is equal to a dollar's worth of programmer labor.

This conclusion makes intuitive sense and can be generalized for any number of inputs [Ref. 5: p. 175]. What the relationship says is that if, at any point, output can be increased by taking a dollar from input X and applying that dollar to input Y, then it is beneficial to do so. The equilibrium point will necessarily be where the ratio of the marginal productivity to cost for all inputs is equal.

2. The Labor Quantity Decision

The second problem is to decide the optimal quantity of an input to utilize in the software development process. Programmer labor will be used as a representative input.

It was shown above that the marginal productivity of programmer labor in the production of software, holding other factors constant, is positive over the relevant range. In other words, an incremental increase in programmer labor will result, up to a point, in an incremental increase in the amount of software produced. The amount of the increase in programmer labor required to produce the incremental increase in software produced is called the marginal input requirement of programmer labor in the production of software (MIRp). If the market price of programmer labor is Pp, then, in order to achieve a marginal increase in software production, a marginal cost (MC) equal to the price of programmer labor multiplied by the marginal input requirement of programmer labor in the production of software will be incurred. The equation is:

$$MC = Pp * MIRp \qquad \text{(eqn 2.5)}$$

or alternately, since it can be shown that the marginal input requirement is equal to the inverse of the marginal product:

$$MC = Pp*1/MPp \qquad \text{(eqn 2.6)}$$

The marginal revenue (MR) received by selling the incremental amount of software produced can also be calculated. If the market price of the software produced is Rs, then the marginal revenue is equal to this market price multiplied by the increase in software produced as a result of an incremental increase in programmer labor. This second term is the marginal product of programmer labor in the production of software. The marginal revenue equation is:

$$MR = Rs*MPp \qquad \text{(eqn 2.7)}$$

Because the flow of funds for costs and revenues occur at different periods in time, it is necessary to discount them to present values before comparing them:

$$\text{Present Value of MC} = Pp*MIPp*e^{-rt} \qquad \text{(eqn 2.8)}$$

$$\text{Present Value of MR} = Rs*MPp*e^{-rt} \qquad \text{(eqn 2.9)}$$

The difference between the present value of the marginal revenue and the present value of the marginal cost is the net present value of a marginal increase in the amount of programmer labor used. If this net present value is positive, that is if the present value of marginal revenue is greater than the present value of marginal cost, then it is profitable to increase the amount of programmer labor used. If the net present value is negative, then it is profitable to decrease the amount of programmer labor used.

At the optimum, all other factors remaining constant, programmer labor (or any other input) should be acquired to the point where the present value of marginal revenue equals the present value of marginal cost. In symbols this is where:

$$Pp*MIRp*e^{-rt} = Rs*MPp*e^{-rt} \qquad \text{(eqn 2.10)}$$

A problem in implementing this type of conceptual framework is the difficulty of developing an accurate production function for the software development process, especially in view of the paucity of good databases on the subject. A major benefit of this type of conceptual framework is its compatibility with linear programming methods as shown by Ein-Dor and Jones [Ref. 6].

III. SIZE OF THE WORK GROUP

A. INTRODUCTION

The complex nature of software development projects has necessitated the decomposition of the overall task into a multitude of lesser tasks and the assignment of groups of people to accomplish those tasks. One would think that the larger the group of people assigned to a task, the shorter would be the completion time for the task. Therefore, in order to meet project deadlines, attempts have been made to speed the completion of complex software development projects by simply adding more manpower to the project. The fallacy of this belief has been widely noted, most prominently in Brooks' widely read book The Mythical Man Month in which Brooks identified some of the factors which restrained increased group size from resulting in decreased project completion time; and in which he described how "adding manpower to a late software project makes it later." [Ref. 3: p. 25] The impact of this phenomenon on the software production function was discussed in the previous chapter. This chapter will analyze how and why the size of the work group contributes to this phenomenon, and how the negative influence on productivity may be mitigated.

B. PRODUCTIVITY IN GROUPS

From studies as well as from our own work experience we know that members of a group working on a task do not spend all of their time doing constructive work. Some percentage of the time is spent on coffee breaks, meetings, illness, training, vacations, communicating, socializing, etc. For a 10 member group "the non-productive time expected for each

member is 25 percent for vacation and the like; 10 percent for idle time; and a base of 10 percent for time spent communicating: a total of 45 percent. We may therefore estimate that 55 percent of each employees time can be considered productive in a group of up to 10 employees." [Ref. 4: p. 3] Fried defines productivity in a software development project as "developing a system with the following characteristics: - Maintainability (documented, modular, etc.) - Effectiveness (meets actual user needs) - Efficiency (uses minimal resources)." [Ref. 4: p. 8]

The portion of non-productive time that is most variable with group size is the communication time. If each member of the group has to interact with each other in the accomplishment of the task, the number of interactions rises dramatically with the number of people involved. If K were the number of people in the group, the number of interactions (N) would be given by the formula:

$$N = K*(K-1)/2 \qquad \text{(eqn 3.1)}$$

This formula shows that the number of interactions in the group increases in exponential fashion with an increase in group size. This communications effort has proven to be a determining factor of productivity time in a group. Fried [Ref. 4] has developed the following formulae for computing the the percentage of productivity time:

$$Pt = K*(T* .55 - .0001*(K* K-1 /2)) \qquad \text{(eqn 3.2)}$$

where:

Pt = productivity time
T = individual employee hours per work period
K = the number of people in the group.

The productivity percentage in the work group is therefore:

$$Pp = 100*(.55 - .0001* K*(K-1)/2) \qquad \text{(eqn 3.3)}$$

where:

Pp = percentage of productive time

K = the number of people in the group.

Solving the above equations for a 10 member group working a 40 hour work week:

$$Pt = 10*(40* .55 - .0001(10* 10-1 /2)) = 218.2$$
$$Pp = 100*(.55 - .0001 10*(10-1)/2) = 54.55$$

whereas for an 80 member group working the same hours:

$$Pt = 80*(40* .55 - .0001*(80* 80-1 /2)) = 748.8$$
$$Pp = 100*(.55 - .0001* 80*(80-1)/2) = 23.4$$

Table I demonstrates how the productivity percentage varies for groups of various size.

Fried [Ref. 4], and Weinberg [Ref. 7] have experienced this inverse relationship between group size and productivity in complex projects with which they have been associated. Furthermore, Fried postulates that it is possible to reach a point of negative marginal productivity. This is consistent with Brooks' [Ref. 3] earlier findings that, after a point, adding manpower can increase time to completion rather than decrease it.

TABLE I

Group Size and Productivity Percentage

Group Size	Productivity Percentage
10	54.55
20	53.1
40	47.2
60	37.3
80	23.4

C. SMALL PROJECT TEAMS

The above findings suggest that, on the basis of productivity time, project teams should be created which are of limited size. A team with two members would seem to have the highest productivity percentage; but the additional coordination and communication that would be required between groups, as well as the limited division of labor possible within the group, would eliminate any possible advantages. Alternately, too large a group results in low or negative marginal productivity.

Brooks [Ref. 3] encountered this dilemma of balancing the desireable aspects of small groups against the absolute need to produce the large and complex OS/360 system within time and budget constraints. He described his problem as follows:

> For effiency and conceptual integrity, one prefers a few good minds doing design and construction. Yet for large systems one wants a way to bring considerable manpower to bear, so that the product can make a timely appearance. How can these two needs be reconciled?" [Ref. 3: p. 31]

The answer that Brooks [Ref. 3], Mills [Ref. 8], and others have advocated is the chief programmer team concept. This concept calls for a 10 person team headed by a chief programmer who designs, codes, tests, and documents the system; and who is totally responsible for the product. All the other team members are tasked with supporting the chief programmer in his duties. The other members of the team and their duties are:

- The "copilot" who serves as the primary assistant and understudy to the chief programmer;
- The administrator who handles the logistics and administrative coordination for the team;
- The editor who reviews the chief programmer's rough documentation and performs the necessary editing and reworking required to produce the final product;
- Two secretaries, one each for the administrator and the editor, for the necessary typing, filing, correspondence, etc.;
- The program clerk who maintains the program product library;
- The "toolsmith" who provides basic utilities, creates macro libraries, and in general facilitates and ensures the adequacy of computer services;
- The tester who designs and plans module and systems testing, produces test cases, test data, etc.;
- The language lawyer who is expert in the chosen programming language and can advise the chief programmer on sophisticated or intricate uses of the language. [Ref. 3: pp. 32-35]

The hierarchy of individuals performing specialized functions in support of a group leader not only provides the benefits of division of labor and specialization discussed earlier, it also provides conceptual integrity in design and coding, as well as simplifying the interpersonal communication required. This reduction in communication requirements coupled with the small size of the team results in a higher productivity percentage for the team. Figure 3.1 illustrates the communication patterns within the chief programmer team. [Ref. 3: p. 36]

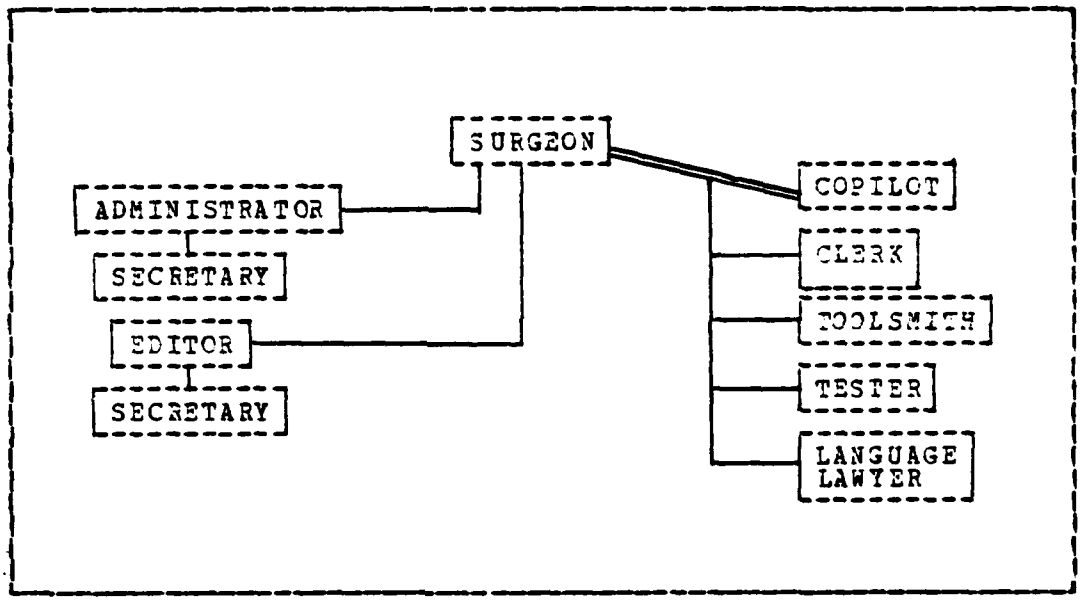


Figure 3.1 Communication Patterns in 10-Man Programming Teams.

1. Social Dynamics Considerations

The small size of these teams and the specialization of function within them also helps to mitigate the negative impact of such social dynamics as the "Commons Dilemma"

[Ref. 9] and "social loafing" [Ref. 10]. These dynamics suggest that individuals in a group may use more than their share of a common resource or contribute less than their share to the common effort if they feel that their excesses or delinquencies will not be distinguishable from the common consumption or effort. The small size of the team and the specialized functions of the team members in the chief programmer team concept alleviate these problems by making each team member accountable for a visible, distinct segment of the group effort.

2. Design Considerations

In order to reap the benefits of small groups such as the chief programmer teams in large, complex projects it is necessary to have many of these teams working concurrently in coordinated fashion. To minimize the coordination and management required, and thereby enhance the productiveness of each team, it is essential that the overall system be designed in a structured, modular manner with clear, unambiguous specifications and documentation. Such standardized design methodologies will be discussed later in the paper. Their benefit is that they allow independent, concurrent production of modules which can be "integrated into the whole without further coordination." [Ref. 4: p. 10]

D. SUMMARY

The above analysis indicates that, in a systems development project, the size of the work groups should be relatively small. The benefits of these small work groups lie mainly in the improved percentage of time spent productively. This benefit results not only from the fewer number of communications within the group; but also from the

ability of small, hierarchically structured groups to mitigate some of the non-productive aspects of group dynamics. Certain managerial and system design techniques may be required to ensure that these benefits result. These techniques include:

- Proper scheduling and task loading based on an understanding of productive time.
- Clear assignment of task and product responsibility, accompanied by measurement and recognition of individual performance.
- Modular design that supports clear assignment of product responsibility. [Ref. 4: p. 10]

IV. STANDARDIZATION OF ACTIVITIES

A. REASONS FOR STANDARDIZATION

Standardization of activities is a very important element of organizational structure because it is the way in which the organization ensures that its efforts will produce predictable results in the quantity, quality, timeliness, and cost of the software produced. An activity is standardized when the procedure is made uniform and consistent.

The advantages of standardization of activities have long been recognized in production processes. In an automobile assembly line the order in which activities are performed, the manner in which they are performed, the qualifications of workers, the rate of production, the tools, parts, etc., are all highly standardized. This standardization is one of the reasons that this type of production is so successful. There are, of course, significant differences between automobile assembly and software development, but the benefits of standardization of activities are recognizeable in both areas.

The goals of standardization are to produce predictable results in quantity, quality, timeliness, and cost. The quantity metric could be lines of code, number of modules, applications programs completed, etc., depending on management's desired control system. The quality metric is a complex and multifaceted one. What constitutes good software is a question that continues to be debated. Reliability, predictability, readability, maintainability, modifiabilty, flexibility, robustness, efficiency, and understandability are some of the concepts currently associated with evaluating the quality of software.

The timeliness and cost metrics are fairly simple concepts. The time it takes to complete a project and the cost of the project will vary with the nature of the project, assigned resources, etc.; with the general goal being to complete the project within the budgeted cost and time period. The predictability of the above metrics is itself a major goal of standardization. From a management viewpoint, the predictability of the outcome of the organization's efforts is absolutely essential for the planning and control of those efforts.

B. SOFTWARE ENGINEERING

The field of software engineering has developed in response to the need to improve and standardize the methods and techniques employed in the software development process. There have been various attempts to define the field of software engineering. Wasserman and Freeman [Ref. 11] defined it as:

> the attempt to seek out and use techniques that can assist in the economical development of software which executes reliably and efficiently on real machines, making effective use of the human resources available. Software Engineering tries to take an overall systems viewpoint in which the optimization of all resources - developmental as well as operational - is considered. [Ref. 11: p. 256]

B.W. Boehm [Ref. 12] shows a slightly different perspective in his definition of software engineering as:

> the means by which we attempt to produce all of this software in a way that is both cost-effective and reliable enough to deserve our trust... (It is) the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them. [Ref. 12: p. 1227]

The decision of which techniques and methodologies to utilize in the software development process is a choice of the technology to employ. This choice is of critical importance to the development process because the technology employed serves to define the software production function. "The production function summarizes the characteristics of the existing technology at a given point in time; it shows the technological constraints that the firm must reckon with." [Ref. 5: p. 146] Therefore, by selecting certain techniques and methodologies, and implementing them as standards for the conduct of the development process, the choice of the technology which will form the boundary of the organization's productivity is made. The importance of structured, modular software design to the implementation and effectiveness of small project teams was discussed earlier in this paper. To provide continuity, design methodologies will be used as the example of how activities in the development process are being standardized to contribute to the success of software development projects.

C. THE DESIGN PHASE

The importance of standardizing the design phase of a software development project has grown with that of the design phase itself. Developing standard design methods is one of the thrusts of software engineering and many approaches have been championed. The standard approaches that have been most widely accepted are those which advocate a structured approach to the design process. The very term "structured" implies that some sort of standard method, mechanism, or approach is used. Stevens, Myers, and Constantine [Ref. 13] have defined structured design as "a set of proposed general program design considerations and techniques for making coding, debugging, and modification

easier, faster, and less expensive by reducing complexity." [Ref. 13: p. 216]

1. Top-Down Design

Perhaps the best known structured design technique is Top-down design which results from a stepwise refinement process. Stepwise refinement is a methodology which consists of the following steps:

1) Start with a high-level, overall statement or description of the desired system function made up of: a) the overall statement of the system function; and b) comments/description of the next level of detail.

2) Refine the above by replacing the comments/description with a) lower level functions; and b) comments/description of the next level.

3) Repeat the refinement until there are no comments left so that the bottom level consists only of functions which can be implemented on the hardware/software machine.

This Top-down design can be represented as a hierarchy of modules in which the "uses" relationship exists between the higher and lower level modules. The "uses" relationship can be interpreted as "requires the presence of a correct version of." [Ref. 14: p 230]

Brooks [Ref. 3] termed Top-down design "the most important new programming formulation of the (1970-1980) decade." [Ref. 3: p. 144] Among the benefits that Brooks attributed to Top-down design were four ways in which it assists the designer to avoid errors or bugs:

First, the clarity of structure and representation makes the precise statement of requirements and functions of

> the modules easier. Second, the partitioning and independence of modules avoids system bugs. Third, the suppression of detail makes flaws in the structure more apparent. Fourth, the design can be tested at each of its refinement steps, so testing can start earlier and focus on the proper level of detail at each step." [Ref. 3: p. 143]

The concepts of modularity and clear structure present in the Top-down design approach are represented in the more resent design approaches although the manner and criteria of the decomposition have varied in some cases.

2. Designing for Change

Parnas [Ref. 15] has proposed a design methodology which focuses on designing software that can be easily changed. His approach uses a modular decomposition based upon information-hiding modules within a hierarchical structure. Parnas [Ref. 15] proposes a design procedure which would include:

1) Identifying all difficult design decisions and those design decisions which are likely to change.

2) Isolating the changeable design decisions into information-hiding modules with clearly defined interfaces which will be unaffected by potential changes.

3) Establish the "uses" relationship between the modules.

4) Set up the "uses" hierarchy by: a) listing the modules at level 0 (i.e. those modules which use no other module); and b) working up the hierarchy to the top level (i.e. that module which is used by no other module).

Parnas' approach to system design has been of significant interest to those in the software engineering field because his methods:

1) Bring software design closer to being a science.

2) Result in programs which are easier to fix and modify.

3) Result in programs which are easily subsettable and extendable.

4) Allow modules to be programmed and tested independently [Ref. 15].

Note that the ability to independently program and test modules which is cited here and in susequent design techniques is what was shown to be neccessary for the effective utilization of small project teams within a large, complex project.

3. Design for Simple Connections and Functional Binding

Stevens, Myers, and Constantine [Ref. 13] have proposed structured design techniques based on principles similar to those of Parnas. These techniques emphasize a structure of simply connected, functionally bound modules. They emphasize the use of structure charts rather than flowcharts in the design phase. Reference 13 provides the somewhat lengthy step-by-step procedure for developing the input-process-output general structure that Stevens, Myers, and Constantine advocate.

The benefits of this design technique include:

1) Its compatability with the HIPO hierarchy charting format [Ref. 16].

2) Better maintainability of resultant programs.

3) Results in independently programmable and testable modules.

4) Ability to identify and optimize critical modules.

5) Ability to develop reuseable modules.

4. Designing Systems as Models

Jackson [Ref. 17] has proposed a different approach to software design. He argues that there are some serious disadvantages to the functional approach to systems design. Among the disadvantages he cites are:

1) The difficulty of applying functional design to complex problems.

2) The frequent requests for changes in system function.

3) The lack of a clear distiction between functions to be performed by software and those to be performed by hardware.

Jackson's approach is to design the system primarily as a model of the reality which it is representing and subsequently superimpose the desired functions on the model. The steps in the process are:

1) Represent each active entity in the real world system to be modeled as a process acting on a dedicated processor.

2) Represent the communication between the processes themselves, and between the processes and the outside world as a data stream.

3) Superimpose desired functions on the model.

Menard [Ref. 18] of the Communications and Computer Science Department of Exxon corporation has used Jackson's design method in combination with top-down implementation and structured walkthroughs. This combination was called the Program Structure Technology (PST). Menard found that

> the benefits derived from the use of PST, measured statistically for several applications, include increased programmer productivity and reduced maintenance costs. With PST as a design method, the programmer can produce double the industry standard number of lines of code per year. The reduced maintenance costs result from encountering fewer bugs in the program code and from having the simpler, structured code which is easier to modify." [Ref. 18: p. 89]

Menard [Ref. 18] found that, whereas the PST method required them to spend more time on the design phase of a project, this time was more than made up in the implementation phase of the project.

D. SUMMARY

The design methodologies discussed above are important for providing conceptually sound frameworks for the design of "good" software; but, in order to realize the maximum benefit, the chosen methodology must be standardized throughout the development project. It is the standardization of the process which provides the organization with the benefits by reducing the necessity for communication and coordination while maintaining design integrity and facilitating successful integration.

The standards themselves form the basis of the organization's planning, control, and evaluation processes. Biggs, Birks, and Atkins [Ref. 19] summarized the importance of standardization as follows:

> The standard approach to phases, steps, activities, and tasks makes it possible to plan, control, and evaluate progress during the systems development process without

> inhibiting the necessary analytical and creative work required to produce successful new systems. The structure allows management to make and monitor incremental commitments and the ability to impact interim results. It is an important key to an organization's effective management of the systems development process. [Ref. 19: p. 47]

The importance of standardizing the activities in the software development process continues to receive management attention. The emphasis on developing standard methods and approaches to requirements analysis, specifications, documentation, integration, and testing manifests the vital importance of activity standardization in the software development process.

V. SUMMARY AND CONCLUSIONS

Organizational structure has long been recognized as having an important impact on an organization's ability to accomplish its objectives. Managers, therefore, need to be aware of how organizational structure affects performance. This paper has provided the managers of software development projects with an analysis of the importance of several elements of organizational structure, and of how they can use this knowledge to make decisions about organizational structure which will have a positive impact on the success of their projects.

The first structural element analyzed was the specialization of activities. It was found that the manager could proceed with specific or general knowledge of the software production function as provided by Brooks [Ref. 3], Friel [Ref. 4], Weinberg [Ref. 7], and Bin-Dor and Jones [Ref. 6], to develop conceptual frameworks to assist in making decisions for optimizing the utilization of the specialized inputs into the software development process. Two of the most important decisions are the determination of the optimal mix of these inputs, and the determination of the optimal quantity of a particular input to aquire.

The optimal mix of the various inputs was shown to exist at that point where the marginal productivity of a dollar's worth of any input in the production of the software output is equal to the marginal productivity of a dollar's worth of any other input.

The optimal quantity of an input to employ, other factors remaining constant, is that quantity at which the present value of the marginal revenue received due to a marginal increment of the input is equal to the present

value of the marginal cost incurred do to that marginal increment.

This type of conceptual framework provides the additional benefit of being compatible with technical analysis and linear programming as shown by Ein-Dor and Jones [Ref. 6].

One element of organizational structure that affects labor productivity and thus the production function is the size of the work group. Brooks [Ref. 3] found that, after a point, increases in programmer labor contributed less and less to software production and that, ultimately, increases in programmer labor would have a negative impact on production. Fried [Ref. 4] experienced similar results as did Weinberg [Ref. 7]. Fried [Ref. 4] and Brooks [Ref. 3] found that communications requirements were the major factor in reduced productivity as group size increased. Fried [Ref. 4] has developed a formula from case studies and practical experience which can be used to calculate the amount of time spent spent productively by a group based upon the size of the group. This formula and Weinberg's [Ref. 7] findings suggest that groups with more than 30 people will spend less than 50 percent of their time doing productive work.

Cass and Edney [Ref. 9] and Latane, Williams, and Harkins [Ref. 10] discovered aspects of social dynamics which also contribute to reduced individual productivity in large groups. These findings indicate that individuals tend to use more resources and contribute less effort if their consumption and performance are felt to be indistinqishable from that of the group.

A possible sclution to the team size problem in view of these findings is the chief programmer team concept. This hierarchically structured, 10 person team is organized in a manner which provides for design integrity, quality output,

simple communication patterns, visible job performance, and high productivity. Successful implementation of this small team concept in complex projects requires a modular project design as well as managerial emphasis on planning, control, and evaluation.

The achievement of good planning, control, and evaluation requires the standardization of software development activities including: requirements analysis, specifications, design, documentation, integration, and testing. The selection and implementation of standard techniques and methodologies represents the choice of technology for the project. This technological choice, in turn, serves to define the production function for the project.

Stevens, Myers, and Constantine [Ref. 13], Parnas [Ref. 15], and Jackson [Ref. 17], among others have proposed software design methodologies which have been used with success as the standard for software projects. The modular structure and clearly defined interfaces resulting from the structured design methodologies allow for the successful division of work among small, efficient programming teams.

Biggs, Birks and Atkins [Ref. 19] emphasize the importance of activity standardization in all phases of the development process as a key element of organizational structure. Its importance is recognized not only because it makes effective planning, control, and evaluation possible; but also for the reduction in communication and coordination it allows.

The field of organizational structure and its impact on organizations' success is vast, with myriad subtle interrelationships. It is an interdisciplinary field with applications from economics, operations research, psychology, sociology, and various technologies. This paper has delved into several of the relationships between elements of organizational structure and the software

development process. A common thread that has appeared in each element is the software development production function. As the database of development projects improves, so too will our ability to analyze and improve the software development process.

LIST OF REFERENCES

1. Stoner, J.A., Management, 2d ed., Prentice-Hall, 1982.

2. Miles, R., Snow, C., Meyer, A., and Coleman, H., "Organizational Strategy, Structure, and Process," Academy of Management Review, p. 546-562, July 1978.

3. Brooks, F., The Mythical Man Month, Addison-Wesley, 1975.

4. Fried, L., "The Impact of Team Size on Systems Development Performance," Auerbach (3-10-19), 1982.

5. Mansfield, E., Microeconomics, 3d ed., Norton, 1979.

6. Ein-Dor, P. and Jones, C., Economics and Management of Computerized Information Systems, Unpublished manuscript, Naval Postgraduate School, 1982.

7. Weinberg, G., The Psychology of Computer Programming, Reinhold, 1971.

8. Mills, H., "Chief Programmer Teams, Principles, and Procedures," IBM Federal Systems Division Report FSC 71-5108, 1971.

9. Cass, R. and Edney, J., "The Commons Dilemma: A Simulation Testing the Effects of Resource Visibility and Territorial Dimension," Human Ecology, p. 371-386: 1978.

10. Latane, B., Williams, K., and Harkins, S., "Social Loafing," Psychology Today, p. 104-110, October 1979.

11. Wasserman, A. and Freeman, P., "Software Engineering Education: Status and Prospects," Proceedings of the IEEE, Vol. 66, No. 8, p. 256-255, August 1978.

12. Boehm, B., "Software Engineering," IEEE Transactions on Computers, Vol. C-25, No. 12, p. 1226-1241, December 1976.

13. Stevens, W., Myers, G., and Constantine, L., "Structured Design," IBM Systems Journal, Vol. 13, No.2, 1974.

14. Parnas, D., "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, p. 226-235, March 1979.

15. Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, p. 220-225, December 1972.

16. "HIPO and Integrated Program Design," IBM Systems Journal, Vol. 15, No. 2, p. 253-257, 1976.

17. Jackson, M., "Information Systems: Modeling, Sequencing, and Transformations," Proceedings, 3rd International Conference on Software Engineering, p. 33-42, 1978.

18. Menard, J., "Exxon's Experience with the Michael Jackson Design Method," Database, p. 88-92, Winter-Spring 1980.

19. Biggs, C., Birks, E., and Atkins, W., Managing the Systems Development Process, Prentice-Hall, 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	LT Kevin Quinn, USN 1360 Atkinson Road Libertyville, Illinois 60048	1